3-5-2020 1:00 PM

Effiicient Computation of Maximal Exact Matches Between Genomic Sequences

Valeria Leticia Portes de Cerqueira Cesar
The University of Western Ontario

Supervisor
Ilie, Lucian
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

Abstract

Sequence alignment is one of the most accomplished methods in the field of bioinformatics, being crucial to determine similarities between sequences, from finding genes to predicting functions. The computation of Maximal Exact Matches (MEM) plays a fundamental part in some algorithms for sequence alignment. MEMs between a reference-query genome are often utilized as seeds in a genome aligner to increase its efficiency.

The MEM computation is a time-consuming step in the sequence alignment process and increasing the performance of this step significantly increases the whole process of the alignment between the sequences. As of today, there are many programs available for MEM computing, from full-text index based algorithms, like essaMEM; to more effective ones, such as E-MEM, copMEM and bfMEM. However, none of the available programs for the computation of MEMs are able to work with highly related sequences. In this study, we propose an improved version, E-MEM2, of the well known MEM computing software, E-MEM. With a trade-off between time and memory, the improved version runs faster than its previous version, presenting very large improvements when comparing closely-related sequences.

Keywords: Sequence alignment, genome, E-MEM, maximal exact matches

Summary for lay audience

To understand relationships between species and the corresponding differences in their DNA, scientists need to use tools such as sequence aligners that can pinpoint these relationships. In some algorithms for sequence alignments, Maximal Exact Matches (MEM) are used as starting points to increase their efficiency. The process is very time consuming and improving the speed in one of its slowest steps benefits the whole alignment performance speed.

There are many programs available for MEM computing, ranging from full-text index algorithms, like essaMEM; to more effective ones, such as E-MEM, copMEM and bfMEM. In this study, we present a second and faster version of the known algorithm E-MEM, E-MEM2. E-MEM2 achieves much better performance when MEMs of highly similar sequences are used, which cannot be accomplished by the other available programs.

Keywords: Sequence alignment, genome, E-MEM, maximal exact matches

Acknowledgments

First, I would like to thank professor Dr. Lucian Ilie for all his contribution and guidance through this journey. This study would not be possible without his thorough feedback, and most importantly his patient and constructive criticism.

In addition, I would like to express my gratitude to my parents and siblings. Thank you for all your support and unconditional love. My gratitude also goes to my "Canadian family", the Stevens, for their encouragements and support.

Finally, I would like to thank my colleague Debanjan Guharoy for always motivating me and cheering me up with his sense of humor; and my lovely boyfriend Ryan Stevens, for always having my back and keeping me going. My appreciation also goes to all of those involved directly or indirectly in providing interest, suggestions and insightful feedback, making this study a reality.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the increase of sequencing technologies and the concept of routine sequencing experiments in laboratories becoming a reality, there is a demand for faster and more efficient algorithms for sequence analysis. The decrease in sequence technology's cost and increase in similar sequenced genomes demands efficient computational technologies for assembly of genomes and comparison of closely related genomes. These advances have contributed to discoveries of new genes, information about gene functions, and identification of variations between related species and individuals.

Maximal exact matches (MEMs) are exact matches between two strings that cannot be extended in either direction, whether towards the beginning or the end of the strings, without allowing a mismatch. The computation of MEMs plays an important role for genome alignments of closely related sequences. MEMs can act as anchor points for genome to genome comparison and as seeds for genome assembly.

A classical approach to compute MEMs between two sequences is with the use of suffix trees (Weiner, 1973). However, this approach requires a large amount of memory and highly engineered modifications into its data structure (Kurtz, 1999). Manber and Myers (1993) introduced the use of suffix arrays as a space-efficient alternative to the suffix trees; and later it was recognized that the enhanced suffix arrays could replace every algorithm that used suffix trees as data structure, with improvement not only in more space efficiency, but also faster and easier to implement (Abouelhoda et al., 2004).

The rise in MEM computing algorithms research occured mainly due to the fact of the increase in popularity in whole-genome alignment programs. One program in particular, MUMmer (Kurtz et al., 2004), based on suffix trees, helped with this increase in popularity, becoming one of the fastest and most efficient software for whole-genome alignment. Changing the indexation of every suffix of a genome sequence to every kth suffix, Khan *et al.* (2009), employed the idea of sparseness for suffix arrays in their MEM computing algorithm sparseMEM. Later, essaMEM (Vyverman et al., 2013) enhanced the sparseMEM algorithm with sparse child arrays, becoming one of the

best programs to compute MEMs between highly similar genomes.

More recently, MEM computing programs with different data structures that do not rely on full text indexes were introduced, increasing the performance while effectively handling memory requirements for large genome sequences. E-MEM (Khiste and Ilie, 2014), with a binary encoding of the genome sequences, samples a subset of seeds, referred to as k-mers, at a fixed step on one genome in a hash table; and all k-mers in the other genome are then compared against the hash table for extensions of the matches found. In Copmem (Grabowski and Bieniecki, 2018), subsets of k-mers in both genomes are sampled using the properties of coprime numbers and a technique based in counting sort. The lastest algorithm, bfMEM (Liu et al., 2019) makes use of a Bloom filter data structure that filters unnecessary indexation of k-mers in both genomes to avoid irrelevant matches. A finer detail into some of the mentioned algorithms will be presented in Chapter 2 after introducing some concepts in molecular biology.

The computation of MEMs is one of the most time consuming steps during the genome alignment process, and although the previously mentioned programs handle the MEM computation efficiently, it still lacks a faster performance when closely related genomes are compared. Consequently, the purpose of this thesis is to present a second version of E-MEM. An improved version, referred to as E-MEM2, that changes the 28 k-mer-limit to 64, using a wider window-array of 128bits. This permits a bigger extension range of the matched k-mers between the two sequences, while also using an improved algorithm for the extensions of the same. The new version also uses the idea of sampling k-mers of both sequences and a checking list that avoids unnecessary extension of redundant matches; having a better performance compared to the previous one. This approach, despite having a drawback in trading-off speed with memory, shows to be specifically better when comparing closely related genomes, performing much faster than the more recent programs. Such improvements in E-MEM2 will be presented and explained in more detail in Chapter 3.

In Chapter 4, we present the datasets used to test E-MEM2 and the comparison with some of the latest MEM computing programs. Finally, we conclude with a brief discussion of the achievements and future directions of the present research in Chapter 5.

Chapter 2

Background

In this chapter we introduce some basic concepts of molecular biology, as well as the computational background needed to understand further concepts and the central problem this thesis addresses. Then, we present some of the leading programs for maximal exact matches computation.

2.1 Molecular biology

2.1.1 Organisms and cells

In biology, a cell is the fundamental unit of life. It contains the essential molecules of which every living organism is composed. Cells can form a complete organism by itself, such as bacteria, or contribute with other specialized cells to form the structure of a multicellular organism, such as humans. All cells have a membrane, a surrounding structure that separates the internal environment, called the cytoplasm, from the external environment. The ability to allow certain substances to travel into the cytoplasm from the outside environment, while also allowing certain substances to exit through the membrane, allows a cell to maintain its health and ability to replicate itself. In addition, cells are separated into two categories, referred to as prokaryotic and eukaryotic cells (shown in figure 2.1). They differ in structure and composition, but most importantly, have different arrangements of their genetic material. Eukaryotic cells are organized by complex structures formed by internal membranes, also referred to as organelles. The most important organelle of eukaryotic cells is the nucleus. It contains all the genetic information the cell needs for growth and reproduction, the DNA.

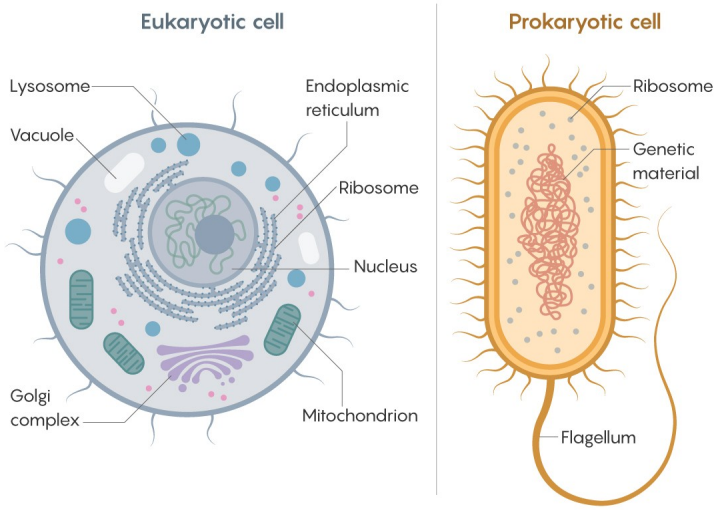


Figure 2.1: An eukaryotic and a prokaryotic cell. (https://www.quantamagazine.org/bacterial-organelles-revise-ideas-about-which-came-first-20190612/)

DNA structure

DNA, abbreviation of deoxyribonucleic acid, consists of complex molecules called nucleotides. Each nucleotide is formed by a sugar group, a phosphate group and a nitrogen base, the latter having four types: Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). The order in which these bases are linked determines the genetic information available for building and maintaining an organism.

The nucleotides are arranged into a strand linking with one another by the covalent bonds between the sugar from one nucleotide and the phosphate of the next nucleotide. The direction of a DNA strand is given by the carbons on the sugar group, where a 5'-end refers to the link with a phosphate and the 3'-end to a hydroxyl. The molecule of DNA is formed when two strands of nucleotides, each running in opposite directions, paired with each other. Hydrogen bonds bind the nitrogen bases of the strands following a specific bonding rule: A bonds only with T, and C only with G. These bonds are called base pairs, and the final structure is a double helix of DNA (Watson and Crick, 1953). Figure 2.2 shows the bonding of the nucleotides and the final double helix structure of the DNA.

The limitations on base pairing are known as complementary base pairing. This way, the sequences of bases on the two strands are related to each other such that the sequence of one stand determines and predicts the sequence of the other. This allows genetic information to be preserved during replication, since each strand serves as a pattern for duplication of the sequence, allowing

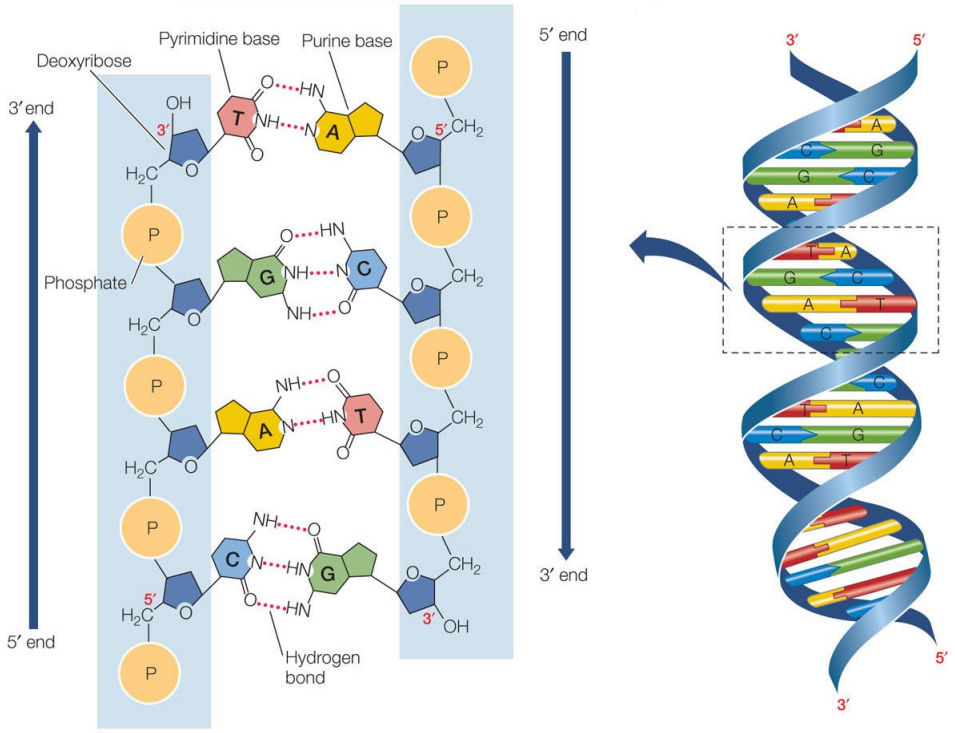


Figure 2.2: DNA molecule twisted into a double helix and representation of its complementary base pairing. (shorturl.at/mvyJU)

the correct transfer of information with an exact copy of the DNA in the original cell.

2.1.2 The genome

The complete set of an organism's DNA is called a genome. The human genome has approximately 3 billion base pairs and to fit into the cell, the DNA is broken down into parts and tightly coiled into 23 pairs of chromosomes. It includes all the genetic instructions a cell needs to maintain life and to reproduce. A unit of DNA that carries information for constructing a specific protein or set of proteins is called a gene. Proteins are responsible for controlling chemical reactions and transfer of signals between cells, and they also make up body structures like organs and tissue. An abnormal protein may be produced in case of a mutation in a cell's DNA. This may cause disruptions in signals between cells and its process, leading to abnormal behaviors. Such misbehaviors can range in severity from a minor effect such as colour blindness to major effect such as abnormal growth in cancerous cells, which can invade and spread to other parts of the body, reducing quality of life.

The human genome is generally the same in all people, with less than 1% of variations in each person's DNA. This small difference is what makes each person different from each other in

appearance and in health (Collins et al., 1997). The idea of studying the human genome and other species' genomes is essential for researchers to better understand the genome itself. Researchers can understand common features between different organisms as well as differentiate genomic elements that are unique in closely related species. Comparing different human genomes can lead to identification of variants in genes with a special role in diseases. The knowledge of how genes and proteins function is essential to impacting the fields of medicine, biotechnology and the life sciences as we learn how to tailor drug therapies for individual needs.

One successful example of a genome study is the human genome project (Lander et al., 2001), which took place from 1990 to 2003. It was an international collaboration to determine, store and render publicly available sequences of almost all the genetic content of the human genome. Sequencing human DNA gives scientists information on which parts of DNA contain genes and which parts carry information on gene regulation. This study contributed to the identification of more than 1,800 disease genes as well the genome information of several other organisms that are important to medical research, such as the mouse (*Mus musculus*) and chimpanzee (*Pan troglodytes*).

2.2 Sequence alignment

The alignment of sequences of DNA, RNA or protein help scientists to identify similar regions with functional, structural or evolutionary relationships between the sequences. It can also make predictions by the detection of similarities between sequences of unknown function and sequences whose function is understood.

Sequence alignments fall into two categories: global alignment and local alignment. Global alignments force the alignment of the two sequences in its entirety, from its beginning to its end, to find the best possible alignment; being suitable for closely related sequences. Local alignments, on the other hand, find local regions with the highest level of similarity between the two sequences. Since it does not consider the alignment of the whole two sequences, local alignments are suitable for more divergent or distantly related sequences.

It is possible to align short sequences by hand. However, computing algorithms are a faster and automatic way to efficiently align numerous and longer sequences.

2.2.1 Dynamic programming

Dynamic programming is a general technique for solving an optimization problem by breaking it down into a collection of simpler sub-problems, solving each one at once and then storing each so-

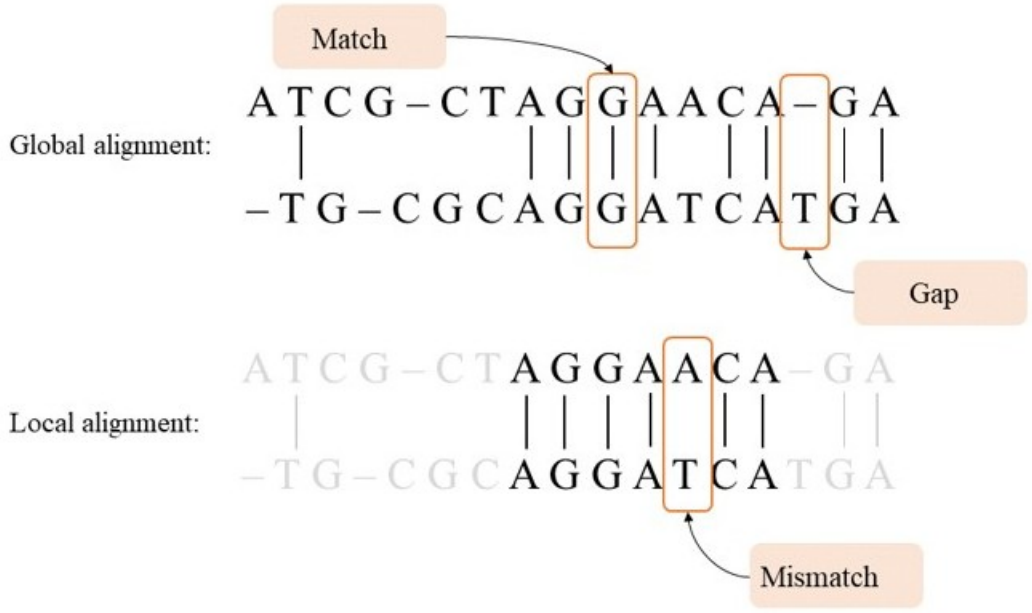


Figure 2.3: Global and local alignment; and representations of a match, mismatch and gap.

lution using a data structure. Next time the same sub-problem occurs, there is no need to recompute it, and instead, one can simply access the previously computed solution.

In sequence alignment, the best possible alignment relies on a scoring scheme where two sequences are given a score on how similar or different they are to each other. For maximal similarity, a positive score is assigned to a match (when two bases are equal), and negative or lower score for a mismatch (when two bases are different) or a gap (spaces introduced in the sequences in order to allow more matches). Figure 2.3 illustrates examples of match, mismatch and gap.

Needleman-Wunsh algorithm

The Needleman-Wunsch is an algorithm (Needleman and Wunsch, 1970) for global alignments. The algorithm relies on the idea that any sub-alignment that tends to a point along the true optimal alignment, leads to the optimal alignment. This can be determined by the merge of and extension of the sub-alignments found. The Needleman-Wunsch algorithm has three main steps:

1. Initialization of the matrix M and score scheme: For a sequence X of length n and a sequence Y of length m, the a matrix with size $(n+1)\,x\,(m+1)$ is required to keep track of the scores assigned. An extra row and column is added, so as to align with a gap. The element $M_{i,j}$ will hold the score of an optimal alignment between the prefix of length i of X and the prefix of length j of Y. Elements in the first row and in the first column of the matrix are initialized with 0, adding to each element of the next row or column the value of the gap penalty. Figure 2.4 illustrates an example of a global alignment for the sequences $X = ATCGAT$ and

$Y = ATCGTA$ using -1 as the gap penalty, 1 as the score for a match and -1 for the score of a mismatch. The following equation defines the terms match, mismatch and gap between sequences X and Y.

$$S(x_i, y_j) = \left\{ x_i = y_j \Leftrightarrow match, x_i \neq y_j \Leftrightarrow mismatch, x_i = - \, or \, y_j = - \Longleftrightarrow gap \right\}$$

2. Filling the matrix: Starting from the upper left corner of the matrix, it is required to know the neighboring scores to find each element's maximal score. From the assumed values, we add the match or mismatch score to the diagonal value; and a gap score to the other neighboring values. Therefore, there are three different values the elements in the matrix can receive, and $M_{i,j}$ is filled with the maximum value. Overall, the equation is referred to as:

$$M_{i,j} = max \left\{ \left(M_{i-1,j-1} + s(x_i, y_j) \right), \left(M_{i-1,j} + s(x_i, -) \right), \left(M_{i,j-1} + s(-, y_j) \right) \right\}$$

   The final score is given by the total in the element on the lower right corner of the matrix.

3. Traceback the residues for accurate alignment: Starting from the bottom right corner of the matrix, we then traceback the alignments based on the source of each score until the upper left cell of the matrix is reached. If a source is given from $M_{i-1,j-1}$ the letters from the two sequences are aligned, whether with a match or a mismatch. If a source is given from $M_{i,j-1}$a gap is introduced into the left sequence X; and if the source is given from $M_{i-1,j}$ a gap is introduced into the top sequence Y.

Smith-Waterman algorithm

The Smith-Waterman (Smith et al., 1981) is a modification of the Needleman-Wunsch algorithm that computes local alignments. Differing from the previous algorithm, the alignment does not need to stretch to the edges of the matrix, it may begin and end internally. All elements in the first row and column are initialized with 0. In addition, the equation for computing the maximal scores is extended with an additional case of 0, excluding possible negative score alignments. The modified equation is defined as follows:

$$M_{i,j} = max \left\{ \left(M_{i-1,j-1} + s(x_i, y_j) \right), \left(M_{i-1,j} + s(x_i, -) \right), \left(M_{i,j-1} + s(-, y_j) \right), 0 \right\}$$

The traceback is executed like the previous algorithm, only this time, the process starts from the cell with the highest score and terminates when a cell with a score of 0 is reached. Figure 2.5

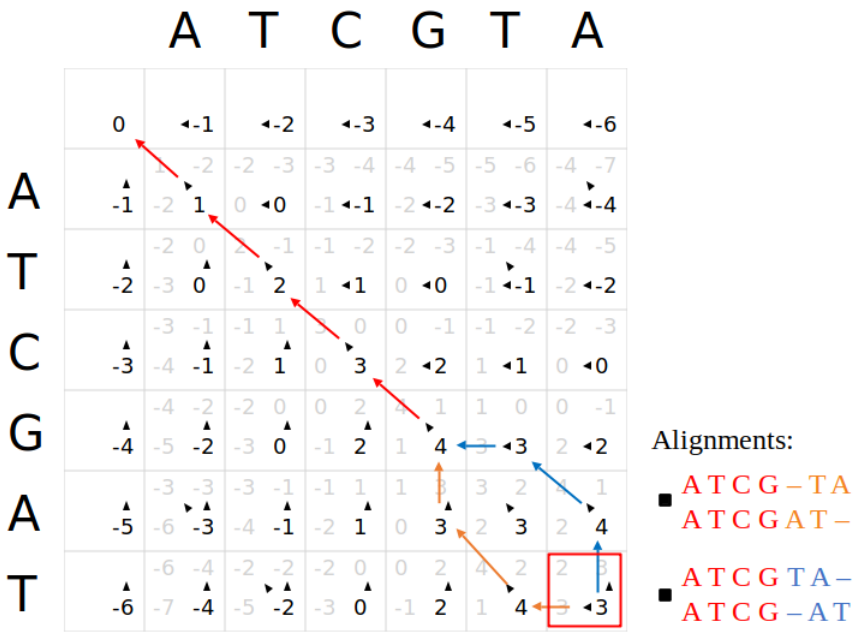


Figure 2.4: Needleman-Wunsh alignment of two sequences. (https://blievrouw.github.io/needleman-wunsch/)

demonstrates the global alignment between two sequences and the possible alignments obtained with the maximal similarity score of 4.

2.2.2 Heuristic programming

A heuristic algorithm is a method for quickly solving problems that are usually time-consuming without the guarantee of finding the best possible solution. It seeks a region of alignment where the alignment score is better than a threshold, typically the expected score between two sequences. A classic example of a heuristic algorithm for alignment of sequences is BLAST (Basic Local Alignment Search Tool) (Altschul et al., 1990). There are several variants of BLAST available, and in the next sub-section we briefly explain the classical approach.

BLAST

The BLAST algorithm generates alignments between a given sequence referred to as “query” and sequences within a database, referred to as “subject” sequences. With the query sequence, BLAST makes a list of 11 consecutive words to be screened in the database for a match. A hit is found

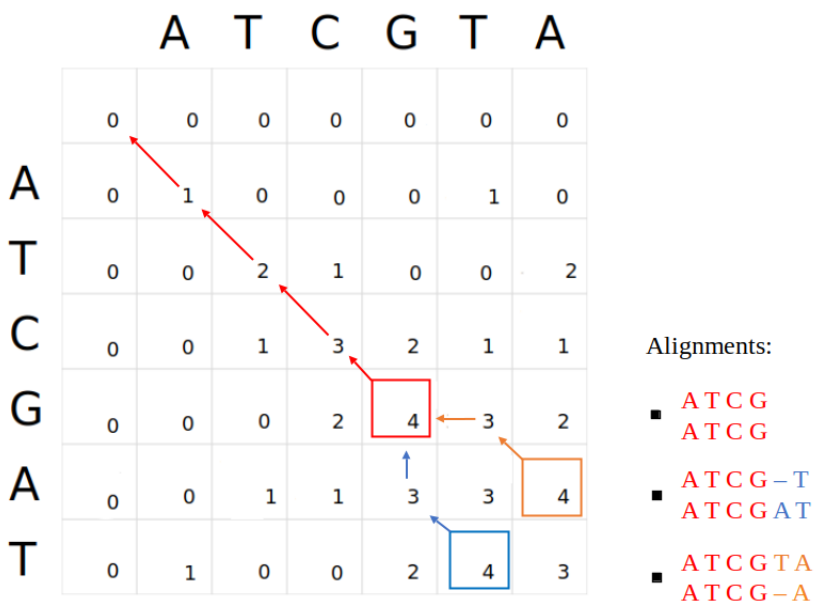


Figure 2.5: Smith-Waterman alignment of two sequences.

Figure 2.6: Illustration of a hit (in red) found by BLAST, which will later extend the consecutive matches to find more similarities.

when 11 consecutive matches are achieved, and from that point the algorithm attempts to extend the similarity in both directions to get sequences with high scores (Fig 2.6). Aligned segments that are above a cutoff score S are then listed in a report for the user together with its score and expected value e. The expected value is a probability of achieving a score S given the query and database size. Generally, the lower the value of e obtained, the more significant the alignment is.

2.2.3 MUMmer

With the the importance of genome analysis along with comparison of entire genomes with one another becoming more pronounced in recent years, there was a need of a new type of program that

could efficiently handle magabase-scale sequences, something that could be achieved by BLAST. As an answer to this need, the institute for genomic research (TGIR) released the first version of MUMmer (Delcher et al., 1999), a system that could perform large-scale genomic comparisons. The algorithm, based on suffix trees, lies on the assumption that input sequences are closely related and starts the alignment performing the computation of maximal unique matches (MUM). A MUM is a subsequence that occurs exactly once in both inputs. If they shared this long exact matching sequence only once in each genome, the MUM is almost certain to be part of the global alignment,allowing for building the alignment around the MUM.

The MUM computing is the first module contained in the software, each subsequent module to perform the alignment of the genomes work independently from each other. The newest version of MUMmer is an open-source system (Kurtz et al., 2004). Thus, researchers can extend or create their own code, making MUMmer increasingly popular. Maximal exact matches (MEMs) can be used successfully as a module instead of MUMs, and in Section 2.4 we will introduce some of the leading programs that can also be used as drop-in replacement in the MUMmer package.

2.3 Text indexing

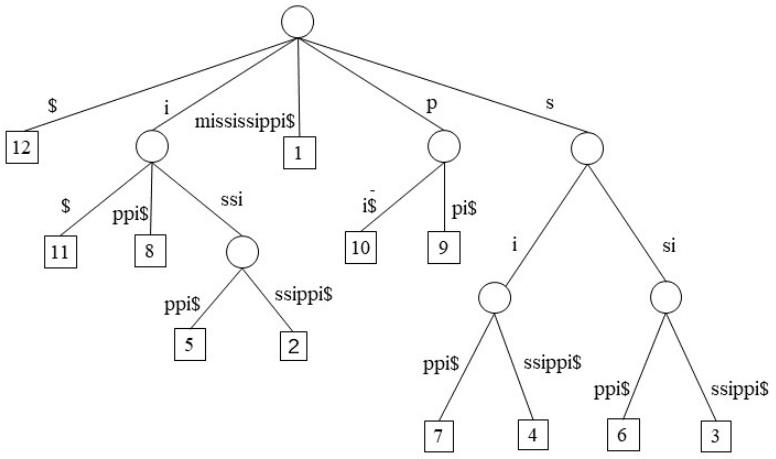
Finding an exact match between two sequences without gaps or mismatches for very long sequences requires some computational work in order to solve problems faster such as the alignments previously mentioned.

Early algorithms to produce MEMs were based on suffix trees and suffix arrays. These concepts are briefly explained in the next sub-section in order to understand the newest approaches for the latest and improved MEM finding algorithms.

2.3.1 Suffix trees and suffix arrays

The suffix tree is a data structure that contains all the suffixes of a given string in a tree-like structure. Suffix trees can be used to solve many string related problems such as matching patterns, finding longest common substrings, and exact string matching.

Given a string S of n characters, if $S = s_1 s_2 ... s_n$, then $S_i = s_i s_{i+1} ... s_n$ is the suffix of S that starts at position i. A suffix tree for the string S is a rooted directed tree with exactly $n + 1$ leaves labeled from 1 to $n + 1$ that represents all suffixes of S. Edges starting out from the same node have string-labels beginning with the different characters in lexicographic order, and except for the root node, every internal node has at least two children (Ukkonen, 1995). To ensure that no suffix is a prefix of another, and therefore, that there will be one leaf for each of the n suffixes of S, a terminal and unique character lexicographically smaller than any of the other characters, \$, is added to the end



1	2	3	4	5	6	7	8	9	10	11	12
m	i	s	s	i	s	s	i	p	p	i	$

Figure 2.7: Suffix tree for $S = mississippi\$$.

of the string. The representation of the suffixes of S is given by concatenating any edge-labels on the path, from the root to the leaf i, spelling out the suffix $s_i...s_n$. Figure 2.7 shows the suffix tree for the string $S = mississippi\$$, the square boxes represent leaves with its respective suffix index.

An example of a problem solved by suffix trees is to find substrings shared by two inputs. Given a string $X = ATCGTA$ and a string $Y = ATCGAT$, we construct the suffix tree for the concatenated string $S = X\#Y\$$, where $\# < \$$. After the suffix tree is built, the longest common substring is given by the deepest node such that its subtree contains leaf nodes from different strings. The figure 2.8 represents the suffix tree of S, indicating in red the path containing the longest common substring; leaves outlined in orange represent indexes that belongs to the string X and leaves outlined in blue for indexes belonging to the string Y.

Suffix link is a well defined feature and important construct of suffix trees. Given a node n with a label az where a is a character and z a non-empty string, then the suffix link of n points to a internal node m with label z. In the case of z being an empty string, then the suffix link of n, $s(n)$, is the root. Suffix links exist for every internal node of a suffix tree (Gusfield, 1997) and can be used to find the initial lcp-intervals for the next suffix position. Thus, the use of suffix link can

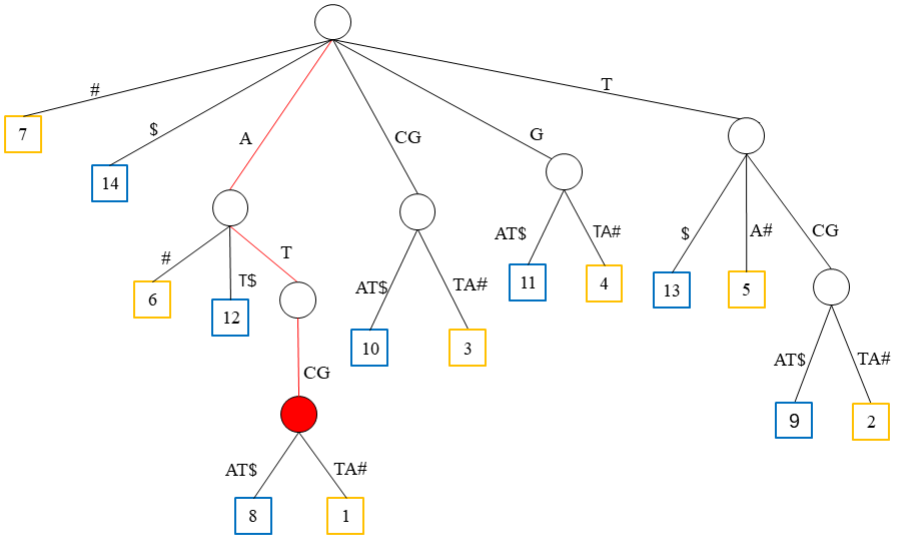


Figure 2.8: Suffix tree for $S = ATCGTA\#ATCGAT\$$

accelerate matching algorithms, since it keeps track of what or how much of the string has been matched so far. Figure 2.9. represents an example of a suffix tree with suffix links.

The suffix array (SA) is an array of integers in the range 1 to $n + 1$, that have the index of the suffixes of S in ascending lexicographic order (Manber and Myers, 1993). The lexicographic order in the alphabet contained in the DNA, for example, is $\$ < A < C < G < T < N$, where 'N' represents an ambiguous base. The table 2.1 shows the suffix array for the string $S = mississippi\$$.

Using the same example previously presented, one could also solve the problem of finding the longest substring between two inputs using suffix arrays. First, as before, we construct $S = X\#Y\$$. For instance, given a string $X = ATCGTA$ and a string $Y = ATCGAT$, $S = ATCGTA\#ATCGAT\$$.

The longest common prefix (LCP) is an array of integers that stores the lengths of the longest prefixes between each pair of consecutive suffixes of a suffix array. Finding the longest common prefix of two neighbouring suffixes, $S_{SA[i-1]}$ and $S_{SA[i]}$ that comes from different input strings, gives us the longest common substring between X and Y. The table 2.2 represents both suffix array and LCP array of S, indicating in red the longest common prefixes. The longest common substring between X and Y has length 4 : $ATCG$.

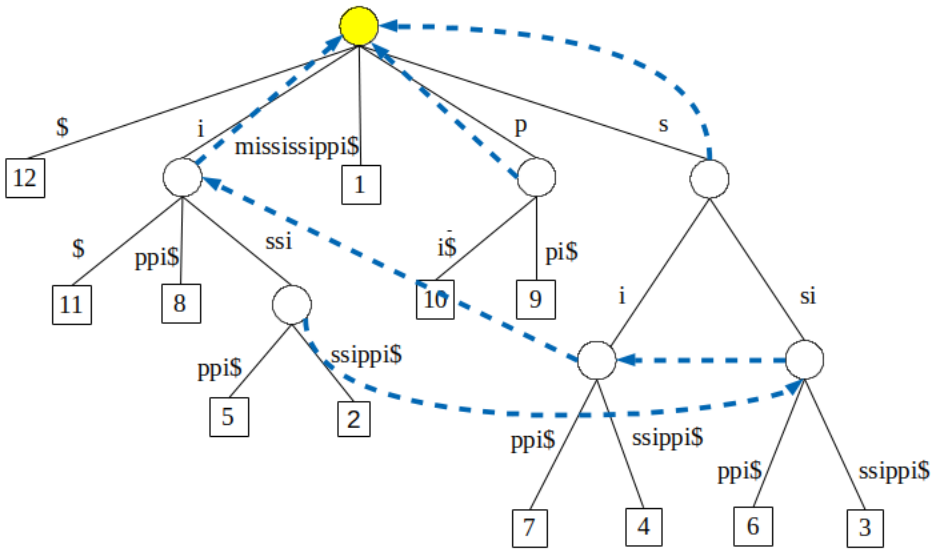


Figure 2.9: Example of a suffix tree of $S = mississippi\$$ with suffix links. The blue arrows represent the suffix links and the yellow node represents the root of the tree.

Index i	Suffix array $SA[i]$	Sorted suffixes $S_{SA[i]}$
1	12	$\$$
2	11	$i\$$
3	8	$ippi\$$
4	5	$issippi\$$
5	2	$ississippi\$$
6	1	$mississippi\$$
7	10	$pi\$$
8	9	$ppi\$$
9	7	$sippi\$$
10	4	$sissippi\$$
11	6	$ssippi\$$
12	3	$ssissippi\$$

Table 2.1: Suffix array of the string $S = mississippi\$$

Index i	$SA[i]$	Sorted suffixes $S_{SA[i]}$	$LCP[i]$
0	13	$	-
1	6	#ATCGAT$	0
2	5	A#ATCGAT$	0
3	11	AT$	1
4	7	ATCGAT$	2
5	0	ATCGTA#ATCGAT$	4
6	9	CGAT$	0
7	2	CGTA#ATCGAT$	2
8	10	GAT$	0
9	3	GTA#ATCGAT$	1
10	12	T$	0
11	4	TA#ATCGAT$	1
12	8	TCGAT$	1
13	1	TCGTA#ATCGAT$	3

Table 2.2: SA and LCP array of $S = ATCGTA\#ATCGAT\$$, indicating in red the common prefixes between the neighbouring suffixes in the suffix array. $ATCG$ is the longest common substring between the two input strings.

A child table (*childtab*) is a structure that contains information of parent-child relationships of LCP of a SA. It can be used to enhance the suffix array and LCP with addition information to determine, for any l-interval $[i..j]$, all its child intervals in constant time. The table containing indexes from 0 to n, contains three values: *up*,*down*,*nextlIndex* defined as follows:

- childtab[i].$up = \min\{q \in [0..i-1] \mid LCP[q] > LCP[i] \text{ and } \forall k \in [q+1..i-1] : LCP[k] \geq LCP[q]\}$
- $childtab$[i].$down = \max\{q \in [i+1..n] \mid LCP[q] > LCP[i] \text{ and } \forall k \in [i+1..q-1] : LCP[k] > LCP[q]\}$
- $childtab$[i].$nextlIndex = \min\{q \in [i+1..n] \mid LCP[q] = LCP[i] \text{ and } \forall k \in [i+1..q-1] : LCP[k] > LCP[q]\}$

2.4 Leading programs

2.4.1 SparseMEM

The sparseMEM (Khan et al., 2009) algorithm is based on the approach of a sparse suffix array (SSA), which consists in indexing every kth suffix of a string, in contrast to a full index that stores every suffix, saving a significant amount of memory. Therefore, if the subset of indexed suffixes are not "too" sparse, the structure can act just the same as a full string indexation. k is called the sparseness factor and the indexation of only a subset of positions trades memory for additional

computation in order to fully find all the MEMs. Therefore, if k is too large, the execution time increases dramatically.

After creating a sparse suffix array of a given string X, the computation of MEMs are done with binary search locating right and left intervals containing the matching suffix of substrings of a string Y, using a top-down approach to SA searching. The algorithm search starts in a position y in Y advancing one character at a time until it narrows down an interval $[s..e]$ in the SA, being s the start index and e the end index, that contains the positions of these exact matches in the string X. The binary search for left and right intervals ends when a mismatch is found.

Using an inverse suffix array (ISA), defined as $ISA[SA[q]] = q$, being q a positive number higher or equal than the length of the string; sparseMEM can simulate suffix links of SA, accelerating the process of moving from one index to another. Details in the algorithm can be find at [Khan et al. 2009].

2.4.2 EssaMEM

The Enhanced Sparse Suffix Array (ESSA) is an improved version of the sparse suffix array and the base for the essaMEM algorithm (Vyverman et al., 2013). It uses the same approach as sparseMEM but presenting two improvements.

The first improvement is the addition of a sparse child array. This method allows a top-down traversal of the hidden suffix tree structure in constant time. The second improvement is the addition of a skip parameter s, that represents the sparseness in the query string. When the parameter s is used, the amount of maximal exact matches and its computation decreases, while increasing the amount of computation to extend those matches.

The default setting of essaMEM features the sparse child array and an optimized value for s. However, this configuration has a weak support for suffix links along with ESSA, having the difficult task of controlling its parameters; and when suffix links are used, s is set to 1. In addition, essaMEM does not support parallelization of the code while using suffix links, but as shown in Chapter 4, the use of suffix links shows a better performance when two close-related genomes are compared.

2.4.3 E-MEM

E-MEM (Khiste and Ilie, 2014) is based on using simple algorithm ideas to produce a more efficient implementation for MEM computation as an alternative for the highly engineered implementation of suffix trees and suffix arrays.

The algorithm indexes a reference string R using a double hashing-based structure that stores k-mers, substrings of size k, along with their positions in R. With a query string Q, each of its

possible k-mers are matched against the previous index structure. The next steps are to extend maximally to the right and to the left all of the matches found. Extensions of a minimum-length L are considered MEMs.

Instead of working with characters in the strings, E-MEM first encodes both R and Q nucleotides with a unique representation of 2-bits: $A = 00, C = 01, G = 10, T = 11$ and stored in arrays of 64-bits; reducing significantly the memory to store a k-mer in comparison with suffix arrays. The k-mers in R are sampled with intervals of length $L - k + 1$, where L is a minimum length of MEM. The idea is that for any MEM of length L or more between R and Q, there must be a k-mer in R starting at a position that is multiple to $L - k + 1$ that is completely contained in the MEM. This avoids unnecessary computation and reduces process time, while ensuring that at least one k-mer of each MEM is stored in the table.

The algorithm uses a 64-bit integer sliding window with a bit mask of $1's$ of length $2k$, shifting its content every 2 bits which samples the next k-mer in the string and then the k-mer is matched against the hash table. Sampling all k-mers in Q ensures all MEMs will be computed, and when a match is found, the algorithm performs its extension phase.

Using a few bit operations that compare all the bits remaining inside the contents of the array containing the k-mer, the algorithm checks for its maximal left and right extension. Since processing the query Q demands the most computation over the whole algorithm, it is possible to use a parameter t that divides the work into threads, decreasing the computational time for this part. E-MEM is able to find MEMs using significantly less memory and time than all previous approaches.

2.4.4 CopMEM

Based on the successful idea of E-MEM, two programs were recently developed: copMEM and bfMEM. copMEM (Grabowski and Bieniecki, 2018) relies on the sampling of k-mers in both strings, given a string R and a string Q, based on properties of coprime numbers. While E-MEM samples every $L - k + 1$ k-mers in R, and Q is sampled with a step of 1 to check the occurrence of matches, copMEM samples both strings with two positive integer numbers, k_1 and k_2, that represents each string sampling step, such that the greatest common divisor between the two numbers is equal to 1 and $k_1 \times k_2 \leq L - k + 1$.

The string R is sampled in two phases, both with a step k_1. First counting the number of occurrences of each hash value and a second time writing the sample positions into appropriate locations of an array, resembling a counting sort algorithm. Q is scanned afterwards with a step k_2 and its k-mers are extracted to later check a match between the positions of the first array, if a match occurs, the algorithm performs the left and right extension character by character.

Given the appropriate choice of parameters, the coprime sampling scheme never misses a seed for a matching window; and to minimize the sampled positions in order to save processing time and computation, k_1 is set as: $k_1 = \lceil \sqrt{L - k + 1} \rceil$ and $k_2 = k_1 - 1$. With a hash table sized set always to 2^{29}, the memory is not as frugal as E-MEM, but its low-level implementation and a strategy of sorting k-mers occurrences of R yields in a much faster performance than the previously mentioned programs.

2.4.5 BfMEM

BfMEM, short for "Bloom filtering for MEM detection" (Liu et al., 2019), introduces the use of a Bloom filter, a probabilistic data structure conceived for membership test with possibility of errors. In bfMEM, the Bloom filter is used to test whether the one string contains a k-mer that belongs to a set of k-mers sampled from a second string. The data structure relies on a bit array B initialized with 0s and a number of h independent hash functions. Every time a k-mer sampled from a given input string Q is added to the bloom filter, an output d is obtained according to the hash functions and represents an index of the array B, where is now set to 1.

After the sampling of the string Q, all k-mers in the string R are scanned; their hash values are computed and tested by the Bloom filter according to the index array B. If any of the tests results in 0, the output is false and therefore, the current k-mer is not a potential for a match. Successful tested k-mers are then extended to the left and right, and considered a MEM if its length is at least L.

The best performance of bfMEM is when it is parallelized over multicore computers. While E-MEM supports the parallelization of processing only the query, bfMEM supports the parallelization of both steps of querying both strings equally dividing them into pieces according to the number of threads used.

Chapter 3

E-MEM2

In this chapter, we introduce an improvement to the E-MEM algorithm. It uses a new approach for sampling the k-mers in both sequences, a bigger sliding window and a better extension algorithm. In addition, the new algorithm makes use of a structure that stores MEMs after the extension to be compared against future MEMs, checking for redundancy and avoiding meaningless computations.

The algorithm requires three input parameters, a reference sequence R, a query sequence Q, and a minimum length for the MEMs to be computed that is set as default to 100. E-MEM2 uses the same algorithm for dividing its input sequences into D pieces and more information about this parameter can be found in (Khiste and Ilie, 2014). We are describing only the case D=1.

E-MEM2 starts by encoding and hashing the reference sequence into a hash table that is 1.75 larger than the number of k-mers in the sequence. This number is estimated by the total amount of bits in the sequence as function of the minimum MEM length *minL* and the k-mer size, set as default to 64. Then, the algorithm encodes the query sequence and its k-mers are sampled to be matched against the reference hash table. Using a list to check for redundant MEMs, relevant matches are extended and MEMs presenting a length greater or equal than *minL* are reported. As E-MEM, E-MEM2 can also run in parallel mode using OpenMP directives. The algorithm 3.1 represents the pseudo-code of E-MEM2.

We describe details in the remaining part of this chapter.

3.1 Sequence encoding

In E-MEM2, the algorithm makes use of a *ASCII* table to convert the characters in the sequences from *char* to *int*. With 128 elements in an asciiArray initialized with 0, specific positions that correspond to decimal values in the letters C, c, G, g, T, t are replaced by $1, 1, 2, 2, 3, 3$ respectively; since the array is initialized with 0, there is no need to replace values for A, a. The sequence is then stored in an unsigned 128-bit array holding 64 nucleotides in each element and follows the

Algorithm 3.1 E-MEM2 pseudo-code

Input: Sequences R, Q, and MEM length $minL$
Output: All MEMs between R and Q of length at least L

```
1: encode R
2: l ← L − k + 1
3: create a hash table for R
4: while l ≤ |R| − k + 1 do
5:     for sv ← 1, s_1 do
6:         hash the k-mer at position l + sv of R
7:     end for
8:     l ← l + L − k + 1
9: end while
10: encode Q
11: while l ≤ |Q| − k + 1 do
12:     get the k-mer at position l in Q
13:     match against the hash table of R
14:     for each occurence of r with extension ≥ L do
15:         currMEM_list
16:         if not in then
17:             add MEMs_list
18:             if (|MEM| ≥ T) then
19:                 add MEM to currMEM_list
20:             end if
21:         end if
22:     end for
23:     l ← l + s_1
24: end while
25: output MEMs_list
```

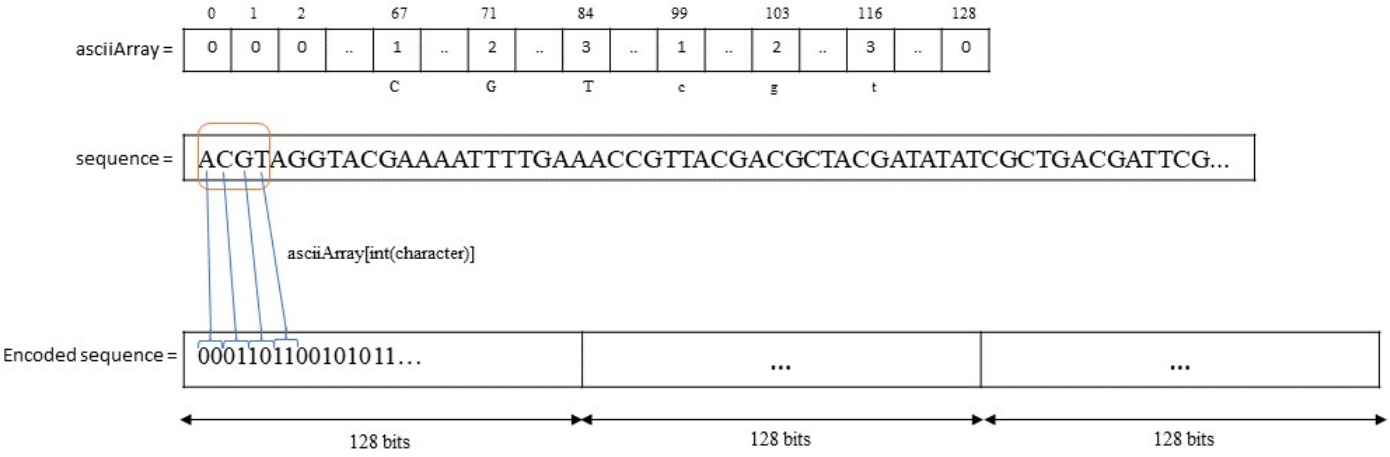


Figure 3.1: Encoding of an input sequence.

same representation as E-MEM $A/a = 00, C/g = 01, G/g = 10, T/t = 11$. An example is shown in Figure 3.1. This method leads to a much better performance in this step as an alternative to the "switch-case" method used in E-MEM.

3.2 Sequence processing

K-mer storage and K-mer search

Using the same technique as E-MEM, we search k-mers from the query sequence for matches against the stored k-mers sampled from the reference sequence. Using the same data structure and hash function as E-MEM, E-MEM2 amplifies its bit representation to accommodate the unsigned 128 bit, allowing the sampling of larger k-mers and a faster comparison of bits in the extension step.

To avoid redundant hits, k-mers are sampled from the reference at interval s. In the E-MEM algorithm, the step s to sample the k-mers is set to $minL - k + 1$, ensuring that at least one k-mer is stored in any possible MEM of length $minL$, reducing memory on the storage of the k-mers. E-MEM checks all k-mers of the query. The new strategy trades memory for speed by indexing more k-mers of the reference while checking fewer k-mers of the query. We increase the number of k-mers in the reference sequence, storing groups of s_1 consecutive k-mers with a step of s_2. We define s_2 as the largest multiple of s_1 that is less than or equal to $minL - k + 1$; and $s_1 = \lfloor \sqrt{minL - k + 1} \rfloor$. With the increase in the amount of k-mers stored, to maintain a high performance, we keep the

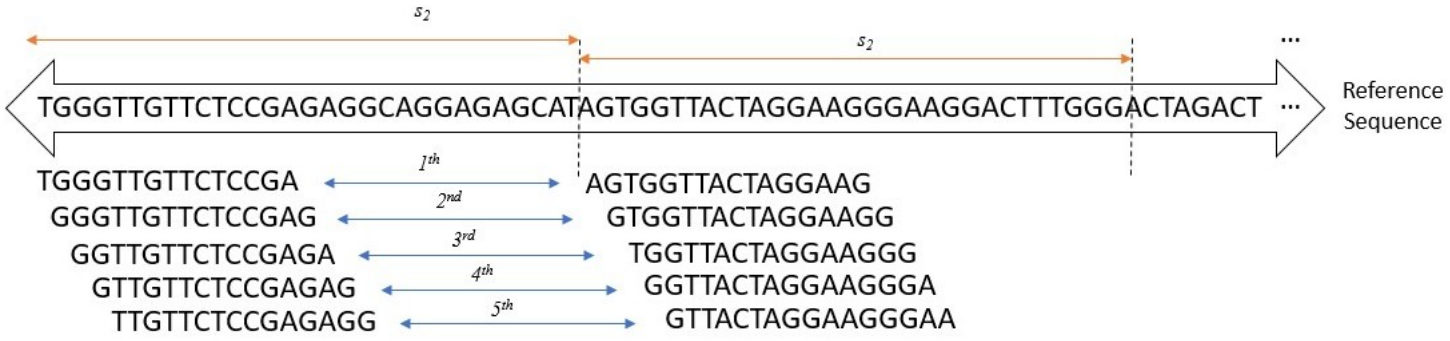


Figure 3.2: Illustration of the k-mer hashing technique when $s_1 = 5$, $s_2 = 30$, storing the shown k-mers of length 15.

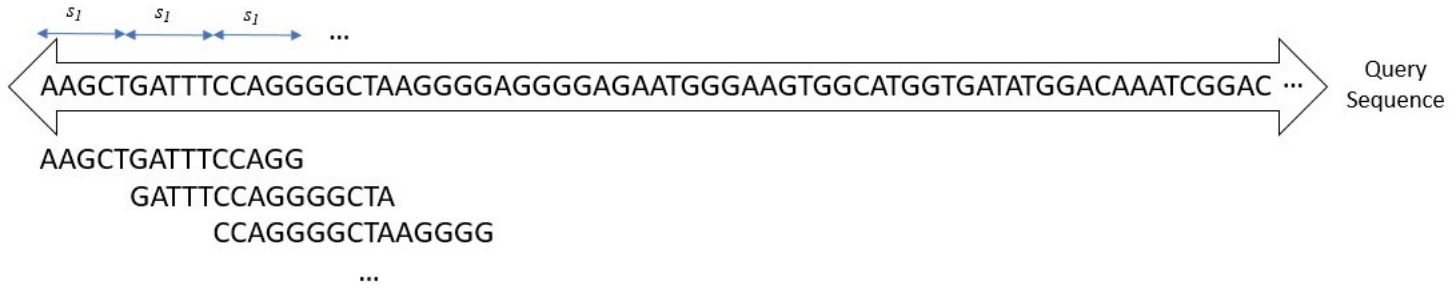


Figure 3.3: Sampling of the query sequence for k-mers of length 15 when $s_1 = 5$, $s_2 = 30$.

load factor under 50% by increasing the size of the hash table by a factor s_1.

Increasing the number of k-mers stored in the hash table makes the sampling of all k-mers in the query sequence unnecessary. In the query sequence, we considered only every $s_1 th$ k-mers to be searched against the hash table for matches and further investigated for a possible MEM. Similar bit operations as E-MEM are performed to extract the k-mers, using a combination of two $2k$ 1's bit mask that represents a 128-bit sliding window that extract the k-mers. Figures 3.2 and 3.3 demonstrate an example of the sampling technique in both reference and query sequences, respectively.

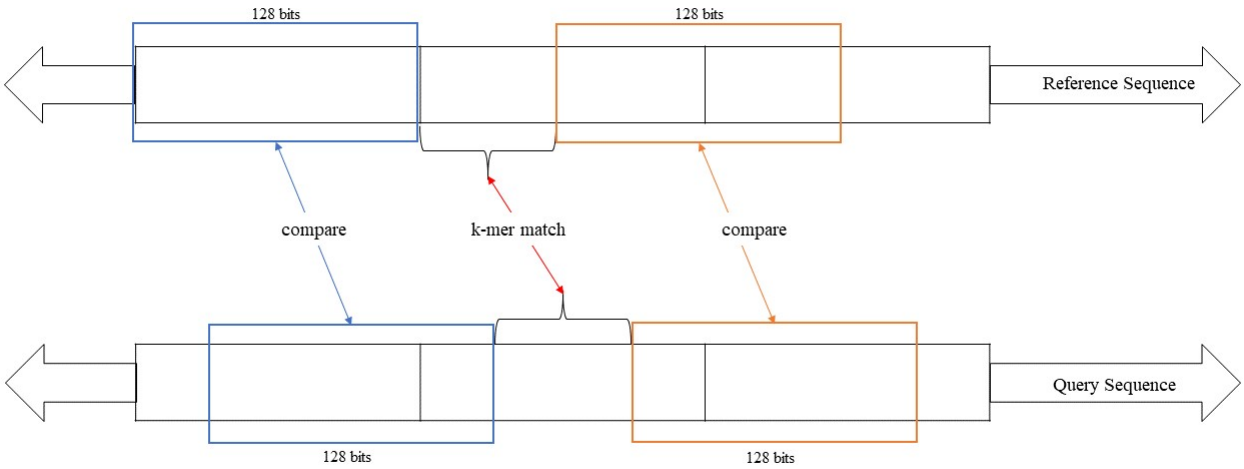


Figure 3.4: *k*-mer matching using blocks of 128 bits.

3.3 128-bit-block extension

Every time a *k*-mer from the query sequence matches a *k*-mer in the hash table, a *k*-mer hit is found and all positions at which it occurs in the reference are investigated for possible extensions. Unlike E-MEM, most of the extensions are made comparing blocks of 128 bits; only if one of the extensions reaches the final 128 bits in the sequence is the extension performed on the bits left in the current unsigned 128bit. Each block of 128 bits to be compared is stored in a temporary variable and the extension is performed using only two comparison operations that determine the length of the match. If the variables match, the extension is continued using the next 128-bit-block, otherwise, the algorithm counts the length of identical bits and stops the extension step.

To investigate each hit for a possible MEM, an extension in both directions is performed and if the hit length is at least *minL*, the *k*-mer hit is considered a MEM. The process is particularly efficient for the discovery of long MEMs.

3.4 Redundant MEMs list

Investigating all MEMs in closely related reference-query genomes is very time consuming, as various MEMs are rediscovered many times. Thus, the extension step in each one of these already existing MEMs is a waste of time. To avoid the computation of possible redundant MEMs we keep track of the relative distance of the current *k*-mer position with respect to the already discovered

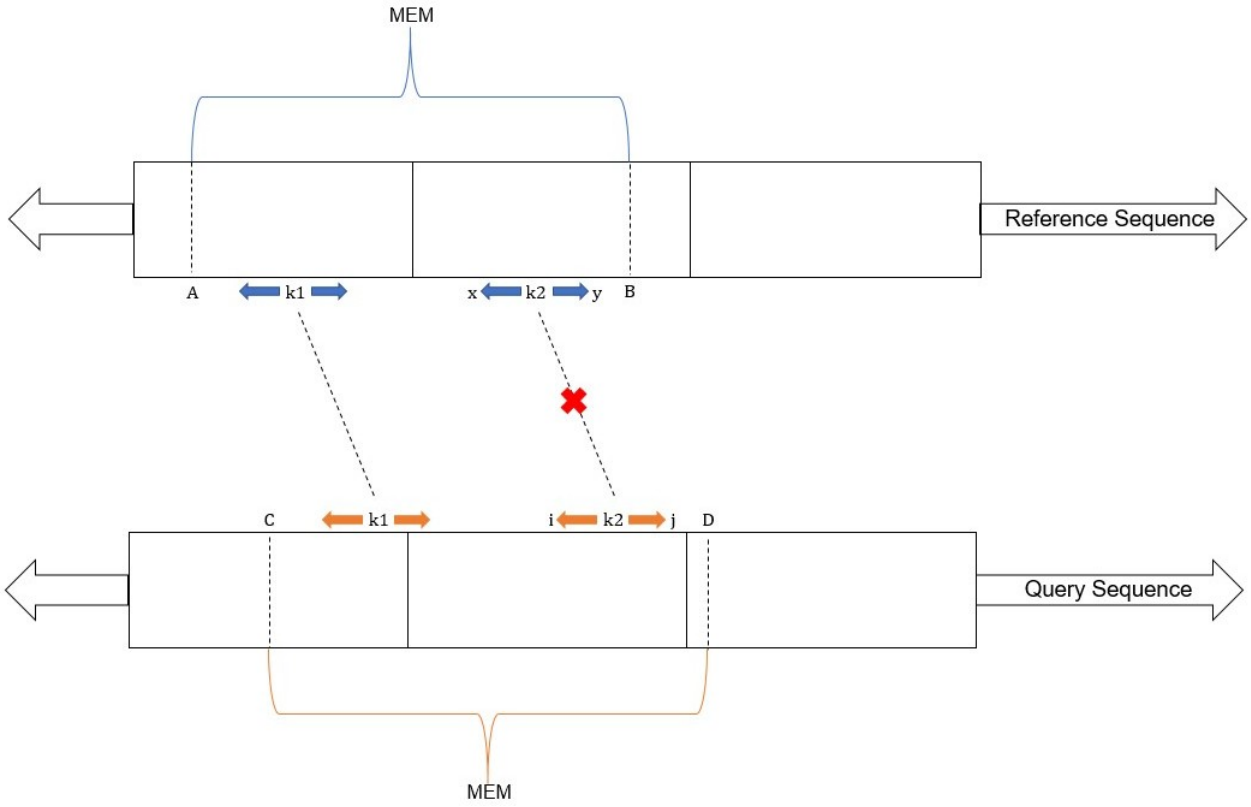


Figure 3.5: Example of a redundant MEM.

MEMs in the query sequence and stored in a linked list structure, here referred to as $currMEMs$. The relative distance is then compared against the current k-mer position in the reference sequence to compute its coordinates. If the coordinates computed are not found in the $currMEMs$, the k-mer is investigated for extensions and therefore, a possible discovery of a new MEM; otherwise the k-mer is discarded.

Given a current position i in the query Q, the MEMs in $currMEMs$ are those whose $b_q \leq i \leq e_q$, being b and e, begin and end of the MEM. Figure 3.5 shows an example of a k-mer that is not suitable for extensions. The figure illustrates one MEM (A, C, l) of length $l \geq minL$ that was discovered by extending the k-mer match k_1 between the query and reference, where A and C are the starting positions of the MEM in the reference and in the query respectively. The match k_2 is contained in an already discovered MEM and should not be extended. To verify the discarded extension computation, the algorithm checks the distance of k_2 from both sides, being $i - C$ on the left side and $D - j$ on the right side, where i and j are start and end positions for k_2. If $x - i = A - C$ and $y - j = B - D$, k_2 rediscovers the same MEM as k_1, thus being discarded.

Optimum length

Adding all current MEMs to the currMEMs list and then checking the entire list is time consuming. Instead, we set a threshold T (default 1000) and then only new MEMs with a minimum size of T are added to the list. The new algorithm has much better performance when the input sequences are highly similar while preserving similar performance for less similar sequences.

Chapter 4

Evaluation

In this chapter we compare the performance obtained by E-MEM2 the top programs for MEM computation: its previous version E-MEM, essaMEM (Vyverman et al., 2013), copMEM (Grabowski and Bieniecki, 2018) and bfMEM (Liu et al., 2019).

4.1 Datasets

Tests were performed using a set of genomes available at the National Center for Biotechnology Information (NCBI) database. We used 4 different human genomes (Hg18, Hg19, Hg37 and Hg514), the sequence of the chromosome 1 from the Hg18, Hg37 and Hg514 genomes, and the genomes from mouse and chimp. The 9 different datasets are indicated in Table 4.1.

4.2 Test setup

All tests were performed on the Shared Hierarchical Academic Research Computing Network (SHARCNET), with using a computer with 12 cores and 256GB of RAM, running CentOS 6.3. Each program was run accordingly to the information given by their authors to generate the correct

Index	Datasets	Organism	Size (Mbp)	Sequences	Reference number
1	Hg18	*Homo sapiens*	3,142	25	GCF_000001405.12
2	Hg18_chr1	*Homo sapiens*	252	1	GCF_000001405.12
3	Hg19	*Homo sapiens*	3,199	93	GCF_000001405.13
4	Hg19_chr1	*Homo sapiens*	254	1	GCF_000001405.13
5	mm10	*Mus musculus*	2,785	66	GCF_000001635.20
6	panTro3	*Pan troglotydes*	3,374	24,132	GCA_000001515.3

Table 4.1: Genomes used for testing.

set of MEMs for the given tests. E-MEM, E-MEM2 and bfMEM can run in either serial (1 core) and in parallel (12 cores) mode. copMEM can only run in serial mode, and essaMEM can only run in parallel when using a sparseness factor and no suffix link support. Since the support of suffix links appears to have a better performance when highly similar sequences are used, we define two settings for essaMEM for comparison purposes: essaMEM-1, with a sparseness factor 8 running in both serial and parallel used for less similar sequences, and essaMEM-2, with suffix link support and no sparseness factor, running in serial only, used for highly similar sequences. The commands used for each program are the following:

- essaMEM-1:
  - Serial: ./mummer -maxmatch -l <100 or 300> -n -k 8 <reference-file> <query-file>
  - Parallel: ./mummer -maxmatch -l <100 or 300> -n -k 8 -threads 12 <reference-file> <query-file>
- essaMEM-2:
  - Serial: ./mummer -maxmatch -l <100 or 300> -k 1 - suflink 1 <reference-file> <query-file>
- E-MEM:
  - Serial: ./e-mem -l <100 or 300> -n <reference-file> <query-file>
  - Parallel: ./e-mem -l <100 or 300> -n -t 12 <reference-file> <query-file>
- copMEM:
  - Serial: ./copmem -l <100 or 300> -o <output-file> <reference-file> <query-file>
- bfMEM:
  - Serial: ./bfmem -r <reference-file> -q <query-file> -o <output-file> -l <100 or 300>
  - Parallel: ./bfmem -r <reference-file> -q <query-file> -o <output-file> -l <100 or 300> -t 12
- E-MEM2:
  - Serial: ./e-mem -l <100 or 300> -n <reference-file> <query-file>
  - Parallel: ./e-mem -l <100 or 300> -n -t 12 <reference-file> <query-file>

We compared results obtained in both serial and parallel for minimum MEM length of 100 and 300. The input sequences used are indicated as Reference vs Query, and programs that could not perform the specified test in the table have the result represented by a dash (-).

Program	Time (*s*)		Memory (GB)	
	Serial	12 cores	Serial	12 cores
essaMEM-1	8,280	1,405	7.21	8.10
E-MEM	1,241	278	3.90	4.00
copMEM	92	-	6.90	-
bfMEM	554	75	4.20	4.30
E-MEM2	416	189	42.80	42.80

Table 4.2: Hg19 *vs* mm10, MEMs of minimum length 100.

Program	Time (*s*)		Memory (GB)	
	Serial	12 cores	Serial	12 cores
essaMEM-1	2,579	478	7.21	8.10
E-MEM	701	197	2.10	2.30
copMEM	47	-	5.20	-
bfMEM	460	56	4.10	4.20
E-MEM2	198	128	17.65	17.65

Table 4.3: Hg19 *vs mm10,* MEMs of minimum length 300.

4.3 Results

4.3.1 Hg19 vs mm10

Tests conducted with the Hg19 and mm10. Tables 4.2 and 4.3 show the results for $minL = 100$ and $minL = 300$, respectively.

For MEMs with a $minL = 100$, there are 537,491 MEMs reported and for $minL$=300, there are 390 MEMs reported in each of the programs.

In both tables, we can observe the improvement of E-MEM2 against E-MEM and the trade-off between speed and space. bfMEM, copMEM and E-MEM2 present similar performances, the last two when using 12 cores. The best performance, however, in both settings, is achieved by bfMEM.

4.3.2 Hg19 vs panTro3

Tests conducted with the Hg19 genome, but with the panTro3 genome as the query input. Tables 4.4 and 4.5 show the results for $minL = 100$ and $minL = 300$, respectively.

There are 132,368,058 MEMs of $minL = 100$ and 951,561 MEMs of $minL = 300$. The very high number of MEMs of minimum length 100 makes both E-MEM and E-MEM2 parallel versions only slightly faster, since most of the computing time is wasted on the post-processing of the MEMs. Still, we can notice the improvement in speed performance in E-MEM2 in both serial and parallel modes. Similar as the previous test, copMEM, bfMEM and E-MEM2 present similar

Program	Time (*s*)		Memory (GB)	
	Serial	12 cores	Serial	12 cores
essaMEM-1	26,640	3,600	7.38	8.28
E-MEM	2,479	1,238	4.00	4.10
copMEM	431	-	7.93	-
bfMEM	1,211	422	5.40	7.70
E-MEM2	1,151	720	42.94	42.94

Table 4.4: *Hg19 vs panTro3,* MEMs of minimum length 100.

Program	Time (*s*)		Memory (GB)	
	Serial	12 cores	Serial	12 cores
essaMEM-1	6,480	1,203	7.43	8.28
E-MEM	1,238	271	2.30	2.50
copMEM	89	-	6.20	-
bfMEM	422	58	3.97	4.20
E-MEM2	245	128	17.80	17.80

Table 4.5: Hg19 *vs panTro3,* MEMs of minimum length 300.

results, with the best performance achieved by the parallel mode of bfMEM in both tests.

4.3.3 Hg18 vs Hg19

Similar to the previous tests, tables 4.6 and 4.7 show the results for $minL = 100$ and $minL = 300$ on tests conducted with the whole human genome, as both reference and query. Here, the tests were run using the annotation genome data Hg18 and Hg19.

For MEMs with a $minL = 100$, there are 159,987,427 MEMs reported and for $minL$=300, there are 750,989 MEMs reported in each of the programs. We can observe a very large margin of improvement between E-MEM2 and E-MEM, achieved by the checking of redundancy of MEMs. Again, the high number of MEMs when the minimum MEM length is 100 makes the post-processing time consuming. However, it is notable that E-MEM2 outperforms the other programs in both tests, and that essaMEM also performs well when inputting highly similar sequences.

Hg18_chr1 vs Hg19_chr1

Tables 4.8 and 4.9 show results of the MEM computation for the input sequences of the chromosome 1 of the genome Hg18 and the genome Hg19. There are 882,480 MEMs of $minL = 100$ and 13,515 MEMs of $minL = 300$.

Analogous to the tests on Hg18 vs Hg19, E-MEM2 and essaMEM are the only programs able to perform under 1 hour, presenting similar results. However, on both tests, E-MEM2 outperforms

Program	Time (*s*)		Memory (GB)	
	Serial	12 cores	Serial	12 cores
essaMEM-2	9,407	-	60.98	-
E-MEM	778,973	108,549	3.73	3.73
copMEM	784,942	-	7.95	-
bfMEM	591,455	91,086	7.62	7.71
E-MEM2	1,387	992	42.67	42.67

Table 4.6: Hg18 *vs* Hg19, MEMS of minimum length 100.

Program	Time (*s*)		Memory (GB)	
	Serial	12 cores	Serial	12 cores
essaMEM-2	3,898	-	60.98	-
E-MEM	209,615	30,612	2.08	2.08
copMEM	203,434	-	5.94	-
bfMEM	269,968	39,251	6.28	6.55
E-MEM2	300	130	17.66	17.66

Table 4.7: Hg18 *vs* Hg19, MEMs of minimum length 300.

essaMEM in both serial and parallel.

4.4 Discussion

E-MEM2 performs significantly better than the original program E-MEM. It has comparable performance with the top programs copMEM and bfMEM. When the input sequences are highly similar, E-MEM2 clearly outperforms all the other programs, to the point of being the only one with acceptable performance.

Program	Time (*s*)		Memory (GB)	
	Serial	12 cores	Serial	12 cores
essaMEM-2	125	-	2.95	-
E-MEM	67,582	7,217	0.30	0.30
copMEM	77,960	-	2.79	-
bfMEM	61,592	51,364	0.97	0.97
E-MEM2	45	21	3.36	3.36

Table 4.8: Hg18_chr1 *vs* Hg19_chr1, MEMs of minimum length 100.

Program	Time (*s*)		Memory (GB)	
	Serial	12 cores	Serial	12 cores
essaMEM-2	123	-	2.95	-
E-MEM	18,498	4,216	0.18	0.18
copMEM	17,049	-	2.57	-
bfMEM	26,300	21,678	0.85	0.85
E-MEM2	20	10	1.41	1.41

Table 4.9: Hg18_chr1 *vs* Hg19_chr1, MEMs of minimum length 300.

Chapter 5

Conclusions and Future Directions

In this study, we considered an efficient solution for MEM computation between large and highly similar genomes. The results show that E-MEM2 performs better than its previous version E-MEM in all tests, and that it is much faster than all programs for MEM computation of very large and closely related genomes, while also preserving a reasonable performance for less similar genomes. In simple words, it improves performance where it matters the most.

Like its previous version, E-MEM2 can also be used as a drop-in replacement for the MUMer software package (Kurtz et al., 2004).

Our main focus was improving the running time, while maintaining reasonable memory requirements. The trade off between time and memory is an important issue for MEM-computing algorithms. It would be interesting to design a variant of E-MEM2 that keeps the memory very low while still providing large time improvements.

Bibliography

Abouelhoda, M. I., S. Kurtz, and E. Ohlebusch, 2004, Replacing suffix trees with enhanced suffix arrays: Journal of Discrete Algorithms, **2**, 53 – 86. (The 9th International Symposium on String Processing and Information Retrieval).

Altschul, S. F., W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, 1990, Basic local alignment search tool: Journal of Molecular Biology, **215**, 403 – 410.

Collins, F. S., M. S. Guyer, and A. Chakravarti, 1997, Variations on a theme: cataloging human dna sequence variation: Science, **278**, 1580–1581.

Delcher, A. L., S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, 1999, Alignment of whole genomes: Nucleic acids research, **27**, 2369–2376.

Grabowski, S., and W. Bieniecki, 2018, copmem: Finding maximal exact matches via sampling both genomes: Bioinformatics, **35 4**, 677–678.

Gusfield, D., 1997, Algorithms on strings, trees, and sequences - computer science and computational biology: Cambridge University Press.

Khan, Z., J. S. Bloom, L. Kruglyak, and M. Singh, 2009, A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays: Bioinformatics, **25**, 1609–1616.

Khiste, N., and L. Ilie, 2014, E-mem: efficient computation of maximal exact matches for very large genomes: Bioinformatics, **31**, 509–514.

Kurtz, S., 1999, Reducing the space requirement of suffix trees: Software: Practice and Experience, **29**, 1149–1171.

Kurtz, S., A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg, 2004, Versatile and open software for comparing large genomes.: Genome Biol, **5**, R12.

Lander, E. S., L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al., 2001, Initial sequencing and analysis of the human genome.

Liu, Y., L. Zhang, and J. Li, 2019, Fast detection of maximal exact matches via fixed sampling of query k-mers and bloom filtering of index k-mers: Bioinformatics.

Manber, U., and G. Myers, 1993, Suffix arrays: a new method for on-line string searches: siam Journal on Computing, **22**, 935–948.

Needleman, S. B., and C. D. Wunsch, 1970, A general method applicable to the search for similarities in the amino acid sequence of two proteins: Journal of Molecular Biology, **48**, 443 – 453.

Smith, T. F., M. S. Waterman, et al., 1981, Identification of common molecular subsequences: Journal of molecular biology, **147**, 195–197.

Ukkonen, E., 1995, On-line construction of suffix trees: Algorithmica, **14**, 249–260.

Vyverman, M., B. De Baets, V. Fack, and P. Dawyndt, 2013, essamem: finding maximal exact matches using enhanced sparse suffix arrays: Bioinformatics, **29**, 802–804.

Watson, J. D., and F. H. Crick, 1953, Genetical implications of the structure of deoxyribonucleic acid: Nature, **171**, 964–967.

Weiner, P., 1973, Linear pattern matching algorithms: 14th Annual Symposium on Switching and Automata Theory (swat 1973), IEEE, 1–11.

Curriculum Vitae

Name:	Valeria Leticia Portes de Cerqueira Cesar
Post-Secondary Education and Degrees:	University of Western Ontario London, ON, Canada 2017 - Present M.Sc. Candidate University of Sao Paulo Ribeirao Preto, SP, Brazil 2011 - 2016 B.Sc.
Honours and Awards:	Western Graduate Research Scholarship 2017 - 2018 Brazilian Scientific Mobility Program Scholarship 2014 - 2015 PIBIC Undergraduate Research Scholarship 2012 - 2014
Related Work Experience:	Teaching Assistant University of Western Ontario 2017 - 2019